Alexander Nassif

Code Description:

In this project, I decided to use Python as my programming language to code an assembler to read through assembly instructions and generate machine code. In my project, I decided to use the built-in module sys in Python to read in arguments from the command line in order to meet the project specifications. I used a dictionary to store all the register values with their corresponding name and register number. For example, register $zero matches to 0, register $a0 matches to 4, and register $t0 matches to 8. I didn't include a functionality for register $0 as the project specifications were a bit vague in what needed to work and what didn't. If you want to test the zero register, you will need to use $zero and not $0.

I did similar methods for the instructions. I first divided the instructions into four categories: R Type Instructions with Shifts, the jr instruction, every other R type instruction, and I type instructions. I divided the R type instructions because the ordering would be different for those, and it made it easier initially to think of the proper algorithms needed when dividing the dictionaries into just dictionaries that followed similar ordering with their machine code. I didn't divide the I instructions because once I had finished the R instructions, I had a good idea of how to format the code and deal with the different ordering of instructions that were the same type. Each dictionary contained either the function or opcode needed to generate the machine code in hex for that specific instruction, so when I wanted to find the opcode or function, I could just look it up in the dictionary.

The main method of my code is the assemble function. The assemble function reads through the *.s file twice. Once it goes through and finds every label that is found and the line of the instruction that comes right after it. This is for finding out offsets. It then goes through again and for every line keeps a count of the lines counting labels (for error lines) and not counting labels (for offsets). It sends out each instruction into either 4 methods depending on which category it falls in from when explained above. These methods format the code and also have error catchers that print out the line number and message that was told for us to print. If there was an error, each method returns "Error", which the assemble function will catch after they are run. If it catches an error, it will end the code. If an I type instruction was called, it will check if it's a branching instruction that has a label to branch to. If it does, it will calculate the offset using the current instruction line and the label lines that were calculated on the first runthrough

Each of the 4 main methods for converting instructions into machine code convert them into binary because I thought it would be easier for me to check my work and visualize. They take the registers from their individual formatting (for example rd, rs, rt) and then convert them into machine code by using dictionaries as lookups for the corresponding register values, opcode, functions, or anything else the individual instruction type needs. It does this in binary. I then made a function to convert them from binary to hex and then once the entire file has run, I output the code into a .obj file of the same name.

For the overall code, for reading in the parameters, I write a for loop that checks the arguments that were passed in. If the file was not found, I print an error, but I continue checking the files because the specifications only say to not generate a .obj file if the instruction inside a .s file was not working right. If one of the .s files was named wrong if multiple were passed, but the rest after them were still right, it will still generate those .obj files.

Build Instructions

The build Instructions for this code are simple since it is a simple python script. I used Python 3.11.9, but most python versions should work for this script. Simply run in cli using your python interpreter and using the cli to input arguments. python3 worked for me, but your specific python command might be different (it depends what type of python is installed on your laptop). For example, a valid run of this code for me was (you may use as many .s files as you want):

```
python3 Project1.py test_case1.s test_case2.s test_case3.s
```

The file should create a .obj file of the same name for each .s file that is inputted. If for whatever reason, this doesn't work. You may go to the bottom of the code, comment out the final for loop (the one that starts with for i in range(1, len(sys.argv))) and call the assemble function inside the code with the filename as the input parameter to also get the code working (although that shouldn’t be necessary as it works on the cli when I tested it).